Towards Document Engineering

Vincent Quint†, Marc Nanard‡, and Jacques André*

† *INRIA/Imag, 2, rue de Vignate, 38610 Gières, France*
‡ *CRIM, 860 rue de Saint Priest, 34090 Montpellier, France*
* *INRIA/Irisa, Campus de Beaulieu, 35042 Rennes, France*

ABSTRACT: This article compares methods and techniques used in software engineering with the ones used for handling electronic documents. It shows the common features in both domains, but also the differences and it proposes an approach which extends the field of document manipulation to document engineering. It shows also in what respect document engineering is different from software engineering. Therefore specific techniques must be developped for building integrated environments for document engineering.

KEY WORDS: software engineering, document engineering, structured editing, integrated environments.

1 Introduction

Software engineering and document manipulation have strong resemblances. Several attempts have been made to use software engineering tools for processing documents. Some of them have been fruitful, others have shown some inadequacies of the tools to the function. In this paper, we try to evaluate the contributions that software engineering concepts and techniques can offer to document processing, but also the limits of these contributions. Because of these limits, we think that specific techniques must be developped for building integrated environments that would allow documents to be produced with all the necessary features. These reflections are based on our experience with different types of systems, especially Grif [Quint86] and MacWeb [Nanard88].

The next section shows the resemblances between the domains of software engineering and document manipulation. Section 3 points out the main differences, considering the semantics of documents and programs, their logical structures, their graphical aspect and the way to use tools for producing documents and programs. Section 4 presents some contributions of software engineering to document manipulation and conversely, and section 5 tries to indicate what an integrated environment would be for professional publishing and desktop publishing.

2 From software engineering to document engineering

Software engineering[1] is concerned with commercial software production (as opposed to programming as an academic exercise or a recreational hobby). In the same way, document manipulation is concerned with the production of real documents, such as technical documentation, books or magazines. In both cases, the writing of small pieces of programs or texts, for personnal use, does not require any software factory or any writer workbench. So, such programs or texts are not relevant of the corresponding domains. On the other hand, small pieces of programs or texts generated during research into program production or during research into writing process are concerned with the present paper.

Document manipulation follows, with a 10 years delay, the same story as programming languages. First, exotic codes (similar to machine languages) were used to drive phototypesetters. Then, some higher level formatters (such as Troff or even TEX) were designed with the same goal as Fortran: to allow machine independency. Going further in abstraction, as Algol 60, Pascal or Ada did earlier, new formatters such as Scribe (and later on, LATEX) used structuration concepts as well as compiling techniques. However, note that WYSIWYG systems have no immediate equivalents in programming languages. At a lower level, one can compare the C language which behaves like a machine independent language and PostScript which becomes a printer independent page description language.

As software engineering is not restricted to the use of programming languages, neither is document manipulation restricted to the use of formatters. Editing large documents requires the same matter as producing large programs. Many of the key words apply to both domains, for instance modularity, configuration management, user interface, reuse, life cycle or integrated environment. Even an aspect such as security (or reliability) is relevant to document manipulation[2]. All of these considerations make it useful to extend to "Document Engineering" what is usually called document manipulation.

Because software engineering is in advance of document engineering, obvious and fruitful proposals have been made to use software techniques and tools for text production. The type and structure concepts are the most commonly used[3]. Among most promising techniques are the methods of stepwise refinement and iterated enhancement, and the use of an integrated environment [Hamlet86].

---

[1] This term seems to have been first employed in 1969 [Buxton69] with a more pragmatic meaning than the one used, e.g., in *IEEE Transactions on Software Engineering*.

[2] A train crash has been quoted as caused by a typographic layout error [André89].

[3] However, note that often so called "structured" editors are actually only "typed" editors: saying in some style sheet that "a section is formatted in 12pt Helvetica, left justified, etc." has nothing to do with a hierarchical structure.

Syntax driven editors, such as *The Synthesizer* [Reps89], Mentor [Donzeau83] or Centaur [Borras87], accept a formal description of the syntax and the semantics of programming languages and handle programs written in the specified language. The abstract syntax allows an abstract representation of the program to be built, from which one or more graphical representations may automatically be constructed. This technique applies similarly to a class of documents. Relevant concepts such as views, annotations, links, gates, etc. fit both program models and text models.

Many other examples of reuse of software engineering techniques by document engineering experts show that this is a good way to follow.

3 Document engineering beyond software engineering

Even if tools for handling documents and programs have much in common, they also show many differences. One of these differences lies in the semantics, which is not the same for documents and for programs. Other differences are related to the logical structure and to the graphical aspect of documents and programs. User interfaces of editors for documents and programs also are very different. All these differences are presented in the next sections.

3.1 Different semantics

From a certain point of view, one can consider programs as a particular type of document. Like other documents, programs are made of lines and characters. They can be edited with text editors and be displayed and printed on the same devices as other documents. But this comparison cannot be continued much further, because the final uses of documents and programs are not the same. A program is intended to be executed by a computer; a document is intended to be read by a human reader, and that makes an important difference.

As stated above, until a certain step in the processing of a document by computer, the analogy between a document and a program is obvious. But, after the document has been printed, there is another step, with no equivalent for programs: the document is processed by human readers, and all the semantics of the document must be accessible during that step.

A document represented in electronic form may not only be formatted and read. It has also often to be processed by several applications. A structured document can be stored in a data base, it can be retrieved easier by information retrieval systems, it can be transformed into several formalisms, it can be reorganized, different forms can be extracted from it. etc. This is one of the reasons why a logical (or abstract) structure is important for a document. If a document is represented in an

abstract form, this abstraction allows other applications to process that document. Because of these many uses of a document, its semantics are not as well defined as the semantics of a program, which is essentially defined for its execution by a computer.

Semantics of a program may be represented in a computer. In addition, it is often elicited by the programmer during the specification phase, before the program is written. It is not the case for documents, except for very rare exceptions.

3.2 Logical structure

Logical structures for documents [Furuta89] are different from those for programs. The syntactical structure of a program is represented by a tree and most syntax driven editors use only this structure, although some structural aspects of programs are not strictly hierarchical, like relationships between modules.

Documents need more complex structures. At a first glance, their logical structure can be considered as a tree: a book contains chapters, a chapter contains sections, a section contains paragraphs, and so on. But documents also use many non hierarchical relationships: all kinds of cross references, indexes, etc. The experience shows that the most interesting features of a document production system like Grif come from the non hierarchical structures. They allow the system to compute and automatically update all numbers, not only section or chapter numbers but also references to sections and chapters. They are very useful for browsing through a document. They allow users to establish relationships between documents or to share some (parts of) documents.

Another type of structure, very important for documents, is represented by hypertexts, where tree structures are less important than non herarchical links. As an example, the primary structure of Concordia [Walker88a] is made of hypertext links.

Documents need more various structures than programs. A tree structure may be used for representing the higher level structure (or primary structure), but some components of a document cannot be represented that way. A table, for instance, is a two-dimensional object that is better represented by a matrix than by a tree.[4] Therefore, several systems use a special structure for tables [Cameron89]. Structured graphics cannot be naturally described by tree structures and need a non hierarchical representation.

The structure of a program is completely defined by a grammar and it must be strictly consistent with that grammar. On the contrary, the structure of a document must tolerate an incomplete definition and must allow some flexibility with respect

[4]Nevertheless, some proposals have been made for representing a table as a tree, but with very strong constraints [Furuta88].

to the model. All types of documents cannot be completely defined in a formal way. Some parts, which are not supposed to be computed, may be less precisely defined.

3.3 Limitations of a logical structure

In a program, the separation between syntactical structure, graphical aspect (sometimes called pretty printing) and free text (comments, identifiers,...) is very clear and stable. In a structured document, logical structure, physical structure and contents can be considered as equivalent to syntactical structure, graphical aspect and free text respectively, but the boundaries between these levels of representation are not so well defined as they are in programs. There are interactions between the three representations [Southall89]. So, a long paragraph, originally written with small characters, is often divided into two paragraphs when it is displayed in large characters: the logical structure is modified for physical reasons. Another example is given by footnotes: in order to limit the number of pages, an author can transform some parts of a paragraph into footnotes.

When defining a generic logical structure for documents, the difference between contents and logical structure is not easy to make. Depending on the document type or the intended use of the document, one can consider a paragraph either as a terminal in the logical structure or as a structured component containing sentences, phrases or words playing different roles (key word, entry in the index or in the glossary, reference to a figure or to a section...). This shows that the boundary between logical structure and contents is not fixed and must be set in accordance with requirements.

A program follows basically one grammar, that which defines the programming language in which it is written. If it contains parts written in different languages, it follows different grammars, but all are of same type. A document follows at least two grammars, very different from each other: the grammar defining its logical structure and the grammar defining the natural language in which it is written. This paper, for instance, follows both the LaTeX grammar (article style) and the grammar of English (it is supposed to...). That implies that a document processing system should be able to handle at the same time computer languages and natural languages, or at least some aspects of natural languages, like spelling, punctuation, style, etc.

For reasons related to semantics (see section 3.1), the process followed by an author when writing a document leads him to often change the logical structure or even to add the logical structure after the contents has been written. This raises problems which are not often encountered when editing programs. Syntax driven editors generally provide parsers for automatically structuring text when it is typed

without structure. The structuring of a document is somewhat different, since it is done manually by the user who indicates the structural components which organize a text that has already been entered in the system.

3.4 Graphical aspect

The graphical structure of programs is usually made up of a sequence of indented lines with some variations of character style for indicating key words, variables or comments. Actually, programs have a linear graphical structure which is divided into lines.

Documents, or at least some of their parts, have more complex, really bidimensional graphical structures. Examples of these complex structures are newspapers, tables, mathematical or chemical formulae and drawings. Formatting languages used in most syntax driven editors usually cannot describe this kind of graphical structures.

As a consequence of the importance of the graphical aspect, document production systems usually have two types of users: a graphic designer and an author. The former is responsible for the graphical appearance of documents, the latter for the logical structure and contents. These two users use the system in completely different ways, and take advantage of different functionalities. In software engineering, a single type of user, "the programmer", is involved in writing a program.

The essential purpose of a document is to be read, what puts a strong emphasis on its visual aspect. The importance of this aspect leads many writers to consider the graphical appearance of a document before its abstract or logical representation. Therefore many document production systems handle a document as a graphic object rather than a logical object.

In order to illustrate the importance of the graphical aspect in a document production system, some figures are given, taken from Grif. The grammar of the language for describing graphical structures is twice the size of the grammar of the language for describing logical structures. In the Grif editor, the part of the code handling the graphical structure is about three times the size of the code handling the logical structure. This indicates that, in a system using a direct manipulation style of interaction, the most important and complex part is dedicated to handling the physical appearance of documents. Manipulating the logical structure, even with a sophisticated model, is much simpler than generating and handling the graphical aspect of a complex document. As programs do not contain the most complex objects encountered in documents, the graphical part of syntax driven editors is not suited for handling these objects.

3.5 The process of writing

Not only are uses of programs and documents different, but so are design methods. Most programs are designed in a top-down manner, with successive refinements. Very few documents can be written that way. Even when the final form of a document must be strongly structured, the author starts often with unstructured text, just for capturing his thoughts. The structuring phase comes later.

On the other hand, some techniques of software engineering may be transposed to documents. Modularity is an example: generic structures of documents can be defined in small, self-contained modules rather than in large monolithic programs. This approach gives to documents the same advantages as to programs: clarity, reusability, maintainability, sharing, etc. Nevertheless, it is difficult to specify modules as rigorously as for programs and encapsulation cannot be as complete [Quint89].

Another approach to modularity is presented in [Walker88b]. Here, modularity is not considered at the generic level, where formal languages are used, but at the specific level: documents themselves are built from modules. That approach has also been taken in MacWeb [Nanard88]. These two examples of modularity in document production systems show that the concepts of software engineering can be transposed in different ways to document engineering.

3.6 User interface

In this section only a direct manipulation interface is considered, as it seems to be the only style of interface users want to use. Concerning the user interface, the basic difference between editors for documents and editors for programs comes from the users themselves. In the case of a program editor, the user is a programmer. That means that he knows about the syntactic structure of the program and that the notion of a tree structure representing the program is familiar to him. Then it is natural to propose commands that explicitly make reference to that structure, for instance for moving across the tree. For a document editor, the situation is completely different. As it must be supposed that the user does not know what an abstract tree is, a different style of interface is needed. The logical structure is useful for enabling the system to make computations on the document, but it must not burden the user with an unnatural model. For that reason, user interface issues in structured document manipulation systems have to be studied further.

Users of systems for the production of structured documents do not only see the user interface under the form of editing commands. They are also faced with the languages that define the logical structure and the graphical appearance of documents. The style of these languages must be adapted to non programmers.

Declarative languages seem to be better accepted by users than procedural ones: they allow the user to express what is required rather than the way to get it. But most users ask for no language at all, at least for describing the appearance of documents. So a graphical language would certainly be the best choice.

Most document production systems use extensively modern user interface techniques, especially direct manipulation. Moreover, many user interface techniques have been developped according to the requirements of these systems. On the other hand, the part dedicated to logical information is often less developped.

4 Cross fertilization

Because of the differences presented in the previous section, tools designed for programs are generally not usable for producing documents, even logically structured documents. It is then necessary to develop tools specifically adapted to documents. Nevertheless, programs and structured documents have common features that suggest that some concepts and techniques used in software engineering could be used in document production too, and conversely.

Programming environments are used for producing not only programs, but also many types of documents related to programs, like specifications, documentations, manuals, helps. All these documents are generally structured according to some well defined model and therefore they are good candidates for structured document editors. Instead of trying to combine in a unique tool all of what is needed for handling programs and documents at the same time, it seems more interesting to use complementary tools: some dedicated to programs, others to documents. That leads to light tools, well suited to the function they have to perform.

Separating programming tools and documentation tools also allows to use the latter in various applications, not only in programming environments. Actually, software engineering is only one of the possible fields of application for structured document systems. Among other fields are technical documentation or specialized publishing (law, medicine, teaching, dictionaries and encyclopediae, etc.).

A typical use of document processing tools in software engineering is the display of complex pictures. Like many document production systems, Grif contains an important component that allows it to display the graphical appearance of a logical structure according to a set of presentation rules. In addition, this component maintains the correspondance between the logical structure and the graphical appearance, so that each change made on one representation is immediately reflected on another. This mechanism has been extracted from the document editor and it is now used by various graphical tools in programming environments. An example is given by the debugging tool presented in [Seze89].

In a programming environment, tools handle abstract objects (in that case the control paths of an ADA program) and use the display component for handling the graphical representation of these objects (trees and graphs in that case) and the interaction with the user on the picture.

A concept of software engineering worth being considered for documents is reusability. In software engineering this concept poses two problems: (1) how to find existing pieces of codes that can be reused in new applications, or even how to know that they exist, and (2) how to integrate these pieces in different contexts. Concerning the first problem, information retrieval tools are available for documents [Salton89], but there is no equivalent tool specifically designed for programs. In documents, a logical structure is a help for solving this problem. The types of components, the attributes (in the SGML sense), the structure itself help to locate the parts that one wants to reuse. In hypertexts, links are also useful for locating pieces of text. Concerning the second problem, the situation is exactly the opposite: programs take advantage of techniques such as object oriented or modular programming. With these techniques, reusing some parts of a program in new programs is made easier. Documents, on the other hand, cannot so easily be reused in different contexts, except for some types of documents, such as technical manuals [Walker88b]. In general, pieces written independently can rarely be merged in a new consistent document without rewriting some parts. But the logical structure may help. Numbers of sections, figure, notes, etc, are computed by the system, based on the logical structure. They can be updated automatically according to the new context. References can also be updated if they are part of the logical structure.

5 Document engineering environments

The previous section has shown that the technology of software engineering is not directly usable for producing documents. This section focuses on some of the functionalities a document engineering environment must provide the user with.

Due to the specificity of document life cycle, the design of such an environment should be task driven. That is a consequence of the non sequential structure of writing, which has been studied in [Hayes80]. For documents, there is no equivalent to the well known specification step, or to the classical sequential approach of software development. Producing a document results in free, non ordered (and often unpredictable) commutations between a set of processes. The most important of them are: producing sentences and paragraphs, gathering ideas and data, reviewing, organizing ideas at semantic level, formatting, logically structuring the document, managing the produced documents.

Another important point the design has to take into account is that the target of a document is twofold: both the reader who perceives it through its visual structure, and the machine which manages the document base and makes its retrieval possible.

Thus, the design of an efficient document engineering environment is today more concerned with integration problems than with the development of new elementary tools for document manipulation.

Environments for professional publishing and environments for desktop publishing have slightly different typical problems. The firsts are concerned with a problem of amount of data, and of long life of these data. The others are more concerned with a problem of user friendliness and of immediate efficiency. Each of these two areas have their own economic interest and scientific problems. In each of them, the amount of produced texts and the number of users concerned make it impossible to ignore their specificity.

An exhaustive list of suitable functionalities for a document engineering environment is of no interest here. Only some of them will be considered. In particular, the importance of structured editors and formatters has already been discussed [Furuta88] and will no longer be considered here.

5.1 Professional publishing

When producing large quantities of documentation (technical documents, price lists, travel brochures, etc.), some part of the information is already available. The most important problems are to access it, to convert it, to adapt it to the context of the intended document and to organize it. Even when a document is produced by a single professional writer, the information handled often has several sources and several authors. For instance, a technical document includes pictures, data collected from databases and shared portions of texts. A given piece of information may be shared by many documents such as preliminary specifications, an implementation reference book and a user's guide.

In that context, information retrieval tools are needed for reusing texts as well as programs. Compatibility between information sources and document engineering environment is also necessary. Standards are the key for this compatibility. Automatic structure recognition is an important complement to document recognition systems. Structure recognition is needed when dealing with old documents which have been produced out of any standard. For instance, most of the documentation of nuclear power plants was produced in the seventies and is unstructured. Even tables have often been typed line by line in a typewriter like style! Restructuring this documentation is an important challenge for making its evolution more efficient.

Flexible approaches make it possible to express at low cost the access path and the structure and type conversions which are needed. As a consequence, the major problems when producing large documents concern the automatic integration of various data sources.

Professional publishing is often a collaborative work. It requires the collaboration of various human expertises (authors, specialists of specific domains, professional writers, typographers, reviewers...). The system is responsible for the ease of communication between these partners and should provide tools for ensuring the consistency of their work.

A document engineering environment should allow the user to "work on" as well as to "edit" a document. Adding and consulting private notes is one of these useful features. A note dynamically linked to some part of a document is fundamentally different from a comment present in a source program. It is a typical problem for hypertext technique. Some software development environments such as HyperCentaur [Vercoustre90] also use hypertext approaches for managing the documentation attached to programs.

Another important part of a document engineering environment should concern the help to the reviewing process too. The most classical of these helps is the spelling checker. Tools such as IBM Critique also provide some simple style checking. The main problem remains that semantics of text are rarely explicit, thus making far more difficult the automatic checking of documents than the checking of programs specifications. Only very specific applications such as deeds produced by sollicitors can take advantage of their strong and explicit semantic. The semantic structure of such documents can be processed and makes their automatic generation possible.

5.2 Desktop publishing

Most desktop publishing systems are today designed for dealing only with the editing and formatting steps. But they actually are being used from the flow of ideas through the production of the camera ready copy. Since desktop publishing users are not professional writers, it is very important both to provide them with simple and efficient tools and to take into account their own natural behavior. A user centered design is required for these tools and their level of integration into a coherent system is the key issue of such environments.

End users rarely are used to deal with abstractions: they "think as they see" and it is not possible to change this fact! This is the reason for the success of WYSIWYG systems. There is, of course, no basic opposition between WYSIWYG and structuration or abstraction. It is just a matter of man-machine interface and of incrementally computing the relevant information to be provided to the user

with a correct feedback [Chen88]. A page editor fully integrated with a structured document editor is still the basic need of most users.

A document engineering environment cannot be a collection of independent tools, but must be integrated as much as possible. The fast, frequent and unpredictable commutation between the various tasks involved when producing a document requires a very fast commutation time between the facilities provided. For instance, the capability of handling personal notes linked to document parts is of no help if the notes cannot be added on the fly when working on a document, or if they cannot be reused by other components.

In document manipulation systems, the friendliness of the user interface is more important than anything else. A feature would not be helpful if the cognitive load for its use is not very low compared with its benefit. It is not surprising that much progress in human computer interaction have been initiated by document manipulation problems. Many scientists working in one area are today interested in the other.

Anyhow, there is no fundamental difference between desktop publishing oriented environments and professional publishing oriented environments. In the long term, the professional document engineering environment will surely focus more on integration and desktop publishing will integrate more concepts and abstractions.

6 Conclusion

Although there are some resemblances between documents and programs, we have put the emphasis on the differences. Pointing out these differences will help in designing new tools, better suited to their role and offering more and more services to users.

References

[André89] J. André, "Can structured formatters prevent train crashes?", *Electronic Publishing – Origination, Dissemination and Design*, vol. 2, no. 3, October 1989, pp. 169–173.

[Borras87] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, *Centaur: The system*, Research Report 777, INRIA, December 1987.

[Buxton69] J.N. Buxton and R. Randell, *Software Engineering Techniques*, Report on a Conference sponsored by the Nato Scientific Committee, Rome, Italy, 27th–31st October 1969, published by Nato, Brussels, 1970.

[Cameron89] J. P. Cameron, *A Cognitive Model for Tabular Editing*, Research Report no. OSU-CISRC-6/89-TR 26, The Ohio State University, Colombus, Ohio, June 1989.

[Chen88] P. Chen, M.A. Harrison, and I. Minakata, "Incremental Document Formatting", *Proceedings of ACM Conference on Document Processing Systems,* ACM Press, December 1988, pp. 93–100.

[Furuta88] R. Furuta, V. Quint, and J. André, "Interactively Editing Structured Documents", *Electronic Publishing – Origination, Dissemination and Design,* vol. 1, no. 1, April 1988, pp. 19–44.

[Furuta89] R. Furuta, "Concepts and Models for Structured Documents", *Structured Documents,* J. André, R. Furuta, and V. Quint, eds., Cambridge University Press, 1989, Elsevier Science Publishers B. V., 1983, pp. 7–38.

[Donzeau83] V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse, and E. Morcos, "Outline of a tool for document manipulation", *IFIP 83,* R. F. A. Mason, ed., pp. 615–620.

[Hamlet86] R. Hamlet, "A Disciplined Text Environment", *Text Processing and Document Manipulation,* J.C. van Vliet ed., Cambridge University Press, 1986, pp. 78–89.

[Hayes80] J. R. Hayes and L. S. Flower, "Identifying the Organization of Writing Processes", *Cognitive Process in Writing,* L. W. Greeg and E. R. Steinberg, ed., Lawrence Erlbaum Associates Publishers, 1980.

[Nanard88] J. Nanard, M. Nanard, and H. Richy, "Conceptual Documents: a Mechanism for Specifying Active Views in Hypertext", *Proceedings of ACM Conference on Document Processing Systems,* ACM Press, December 1988, pp. 37-42.

[Quint86] V. Quint and I. Vatton, "Grif: An Interactive System for Structured Document Manipulation", *Text Processing and Document Manipulation,* J.C. van Vliet ed., Cambridge University Press, 1986, pp. 200–213.

[Quint89] V. Quint and I. Vatton, "Modularity in structured documents," *Woodman'89,* J. André & J. Bézivin, eds., Bigre num. 63–64, IRISA, Rennes, May 1989, pp. 170–177.

[Reps89] T. W. Reps and T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors,* Springer Verlag, New York, 1989.

[Salton89] G. Salton, *Automatic text processing,* Addison-Wesley, Reading, Mass., 1989.

[Seze89] P. de Seze, C. Bonnet, J.-F. Caillet, and B. Raither, "A Graphical Trace Analysis Tool for Ada Real-Time Embedded Systems", *Proceedings of the Sixth Washington Ada Symposium,* Washington D. C., June 1989, pp. 47–52.

[Southall89] R. Southall, "Interfaces Between the Designer and the Document," *Structured Documents,* J. André, R. Furuta, V. Quint, eds., Cambridge University Press, 1989, pp. 119–131.

[Vercoustre90] A.-M. Vercoustre, "Structured Editing—Hypertext Approach: Cooperation and Complementarity", *EP90,* R. Furuta ed., Cambridge University Press, (these proceedings), 1990, pp. 65–78.

[Walker88a] J. H. Walker, "Supporting Document Development with Concordia," *Computer,* vol. 21, no. 1, January 1988, pp. 48–59.

[Walker88b] J. H. Walker, "The Role of Modularity in Document Authoring Systems," *Proceedings of ACM Conference on Document Processing Systems,* ACM Press, December 1988, pp. 117–124.